CS31 (Algorithms), Spring 2020 : Lecture 5

Date:

Topic: Smart Recursion

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

This lecture marks the start of the module on *Dynamic Programming*. Dynamic Programming is an essential tool in the algorithm designer's repertoire. It solves problems that a first glance seems to suggest are terribly difficult to solve. For example, take Problem 2 in PSet 0 — most of you realize brute-force will take way too long, and have been befuddled as how to proceed. But Dynamic Programming will solve it pretty efficiently.

Dynamic Programming, unfortunately, is also something that some students have trouble understanding and using. I know, because I did. But it *is* a very simple idea, and once one gets used to it[1], kind of hard to muck up. We will begin DPs in earnest from next class, but today we explore the main idea behind dynamic programming: *recursing with memory* aka *bottom-up recursion* aka *smart recursion*.

1 Fibonacci Numbers: Recursion with Memory

Let us recall Fibonacci numbers.

$$F_1 = 1, F_2 = 1, \qquad \forall n > 2, F_n = F_{n-1} + F_{n-2} \tag{1}$$

Here is a simple computational problem.

<u>FIBONACCI</u>
Input: A number n.
Output: The nth Fibonacci number, F_n.
Size: n.

Remark: *"Now hold on," I hear you cry, "we are back to handling numbers as input. Then if the input is n, shouldn't we consider the number of bits in n, that is $\log n$, as the size and not n?". Perfectly valid question. The answer lies in the output. The exercise below shows that the number of bits required to write F_n is $\Theta(n)$. This is the reason we are going to take n as the size.*

But I ***am*** *going to cheat a bit below – when I add two numbers, I am still going to* wrongly *assume they are $O(1)$ time operations. In fact, they may take $O(n)$ time since the numbers can be $O(n)$-bits long. However, addition is a much, much faster operation than say running a for-loop, and we are going to just assume addition is an elementary operation.*

✍

Exercise: *Prove that for any n, we have $2^{n/2} \le F_n \le 2^n$. Use induction.*

The definition (1) of Fibonacci numbers implies the following recursive algorithm.

[1] 20 problems should do it

```
1: procedure NaiveFib(n):
2:     if n ∈ {1, 2} then:
3:         return 1
4:     else:
5:         return NaiveFib(n − 1) + NaiveFib(n − 2)
```

The above is a disastrous thing to do. However, for all the problems we will encounter for Dynamic Programming, if you have obtained the disastrous algorithm as above, we are actually probably close to victory. That is, if we have been able to express your algorithm as a recursive algorithm like the one above, then the fix for the above algorithm will probably work for our recursive algorithm as well. More precisely, we are going to see below how to smartly implement the recursion in (1). And this trick, almost always, will also *mechanically* work for the recursion you have obtained.

What is the running time of NaiveFib? As warned before, I am assuming (wrongly) that the addition in Line 5 takes $O(1)$ time. Alternately, you can just think of $T(n)$ as the number of additions required to calculate the Fibonacci numbers. We see that the recurrence which governs the running time is

$$T(n) \leq T(n-1) + T(n-2) + O(1)$$

Even if we ignore the $O(1)$ in the RHS above, we see that the recurrence governing $T(n)$ is eerily similar to (1). And that is not good news – it says $T(n)$ can be as large as F_n which we know from the exercise above is $\geq 2^{n/2}$. Yikes!

First we observe *why* the above algorithm is so time consuming. To see this, let us draw out the "recursion tree" when NaiveFib is indeed called on a certain value. Figure 1 shows the case when called with $n = 6$. We see that many problems are solved *again and again*, recursively. There are two ways to fix this.

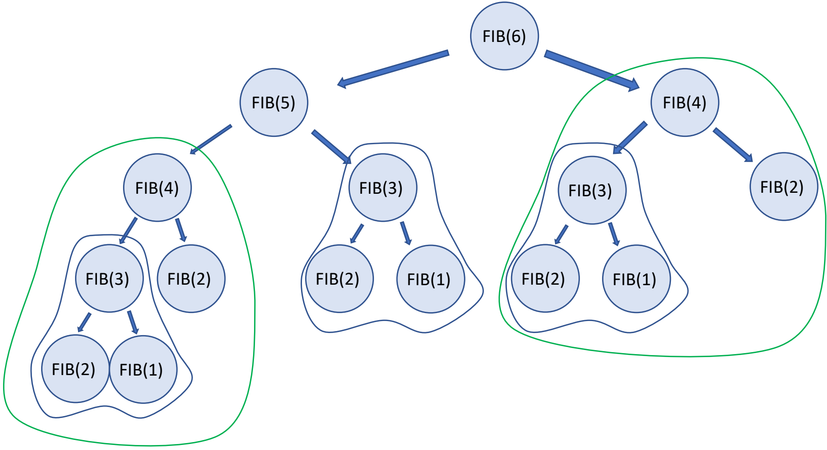

Figure 1: Call of NaiveFib on $n = 6$. The encircled parts of the tree show repeated computation.

Both involve the same principle. The idea is

If we remember the solutions to smaller subproblems,
Then there's no need to take the trouble to re-solve them.

Hey, that rhymed!

Implementation via Memoization.

```
1: Implement a "look-up table" T.
2: Define T[1] ← 1; T[2] ← 1.
3: procedure MemoFib(n):
4:     if T[n] is defined then:
5:         return T[n]
6:     else:
7:         t ← MemoFib(n − 1) + MemoFib(n − 2)
8:         Set T[n] ← t
9:         return t
```

The *memoization* approach gets to the heart of the problem described above. It stores all previously computer Fibonacci numbers in a look-up table T. Different programming languages have different implementations of the look up table; we assume (perhaps wrongly) look ups take $O(1)$ time. Therefore, the running time of the above psuedocode is $O(n)$.

Bottom-Up Implementation: The Table method. This is the method which makes computation more *explicit* and will be what we will use throughout the course. The method described below seems a little wasteful implementation of the memoization idea; it has the benefit of (hopefully) being clearer.

Remark: *Few comments. The table method solves a lot more; for instance, the table method will return all the F_j's for $1 \leq j \leq n$. The memoization method only solves what is needed. On the other hand, memoization includes recursion and implementing a look-up table. Although we have considered these to be $O(1)$-operations, in practice, depending on the system in which it is implemented, the running times can differ. All in all, the table method is more clear cut and easier to analyze. However, when faced in the "real world" do not forget memoization. In any case, if you do use memoization or bottom-up, the important thing to remember is to be correct.*

We first observe that the *solution* to the problem Fib(n) depends on the *solutions* to the problems Fib($n-1$) and Fib($n-2$). Thus, if we stored these solutions in an array (sounds very much like a look-up table) $F[1:n]$, then the following picture shows the dependency of the various entries.

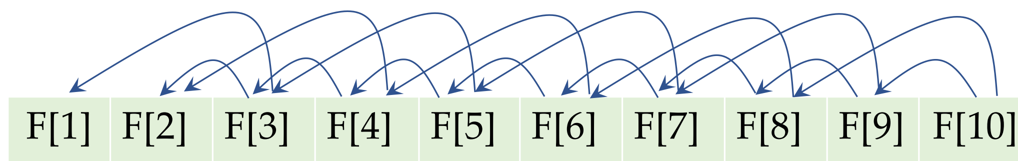

Figure 2: Dependency of the Fibonacci Array

Note two things: (a) The "graph" has no cycles. Otherwise, there will be a cyclic dependency and it doesn't make sense. (b) There are "sinks" in this graph: points from which no arrows come out. These are the *base cases*; $F[1]$ and $F[2]$ don't need any computation; they are both 1. Given this graph above, the algorithm to obtain the nth position $F[n]$ becomes clear: traverse it from the "sink" to $F[n]$. Here's how.

```
1:  procedure FIB(n):
2:      Allocate space F[1 : n]
3:      Set F[1] ← 1; F[2] ← 1.
4:      for i = 3 to n do:
5:          F[i] ← F[i − 1] + F[i − 2]
6:          ▷ Note: the computation of F[i] requires precisely the F[j]'s the F[i] points to
7:      return F[n]
8:      ▷ Note: we have actually found all the F[j] for j ≤ n. The Table method often does more
    work than needed.
```

Exercise: *Implement both the above methods, via memoization and the bottom up table method, in Python 3, and use the* `time()` *routine to see which is faster. Discuss your findings on Piazza.*

2 Binomial Coefficients

Here is another example: *binomial coefficients*. Given non-negative integers $n, k \leq n$, one uses $\binom{n}{k}$ to denote the number of ways of choosing k distinct items from n distinct items. The following is a well known recurrence *equality* (just like Fibonacci numbers) called Pascal's identity.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{2}$$

Of course, it needs base cases. Here there are two. The first is that $\binom{m}{0} = 1$ for all integers $m \geq 0$; there is exactly one way to choose 0 things out of m things: do nothing. The second is that $\binom{m}{j} = 0$ if $j > m$; there is *no* way you can pick more things out of less things. Again, all this you have seen in CS30. Now to the problem at hand.

BINOMIAL
Input: Numbers $n, k \leq n$.
Output: The Binomial Coefficient $\binom{n}{k}$
Size: n.

Remark: *If you remember CS 30, then you know there is an* explicit *formula for the binomial coefficients. But that includes multiplication and division; the above recursive algorithm will need only addition. More importantly, see UGP3 Problem 1 for a very related application without a closed form formula.*

Once again, there is a naive recursive algorithm which the above recurrence (2) readily defines.

```
1: procedure NaiveBinom(n, k):
2:     if k = 0 then:
3:         return 1.
4:     else if k > n then:
5:         return 0.
6:     else:
7:         return NaiveBinom(n − 1, k) + NaiveBinom(n − 1, k − 1)
```

Again, if you write the running time of it using a recurrence inequality, you will see that the time taken is at least the value of the binomial coefficient. How large can they be? Do you remember? Can be pretty large. Rule of thumb $\binom{n}{n/2} \approx \frac{2^n}{\sqrt{n}}$. So no way this will compute for $n = 100$.

But, we can fix this *just* like we fixed NaiveFib. There is one difference — the function above takes *two* parameters (n and k). And thus, our table needs to be *two dimensional*. Do you want to attempt this before reading on? It would be very instructive.

We observe that to compute $\binom{n}{i}$ we need only $\binom{n-1}{i}$ and $\binom{n-1}{i-1}$. Thus if we have "smaller" binomial coefficients, then we can use them to get larger binomial coefficients. This suggests we store for all $1 \leq m \leq n$ and all $1 \leq j \leq k$, the binomial coefficients $\binom{m}{j}$; this we store in a two-dimensional table $B[m, j]$. Once we make this decision, the following code tells us how to "fill up the table".

```
1: procedure Binom(n, k):
2:     Allocate space B[0 : n, 0 : k].
3:     Set B[m, 0] = 1 for all m ▷ Base case of (m choose 0) = 1 for all m.
4:     Set B[m, j] = 0 for all j > m. ▷ Base Case : there is zero ways of choosing a larger number of
       items from a smaller number of items.
5:     for m = 1 to n do:
6:         for j = 1 to k do:
7:             B[m, j] = B[m − 1, j] + B[m − 1, j − 1].
8:     ▷ Note: the computation of B[m, j] requires B[m − 1, j] and B[m − 1, j − 1] and they have
       been computed before.
9:     return B[n, k]
10:    ▷ Note: we have actually found all the B[m, j] for m ≤ n, j ≤ k. The Table method often
       does more work than needed.
```

The running time and space of the above is $O(nk)$ time. Way better than the recursive algorithm and we can easily go for $n = 100$ and $k = 50$ (why don't you code it and see?)